WSO2 REST APIs Design Guidelines

Frank Leymann
Joseph Fosenka
Sanjeewa Malalgoda
Nuwan Dias
Sameera Medagammaddegedara
Malintha Amarasinghe

EXPOSEE

This document has evolved as part of numerous design sessions and reviews of the REST APIs of WSO2 API Manager and WSO2 Enterprise Store Publisher; also, design session on the REST API of WSO2 Machine Learner have been based on these guidelines. A lot of discussion went into the attempt to create a document that may serve as a guideline for other product teams to build a REST API for their products. The ultimate goal of this document is to help creating APIs of a common look-and-feel of all WSO2 products.

WSO2 REST APIs Design Guidelines

frank@wso2.com[1]

Abstract: *In this document we sketch major guidelines to design APIs that comply to the REST architectural style. More precisely, we focus on specifying RESTful APIs over HTTP/HTTPS. These guidelines have been followed to specify the REST APIs of several WSO2 products.*

1 Introduction

When following the guideline of this document the resulting API will reach level 2 of the Richardson Maturity Model ([1], [4]). That means a resource model has been provided, use of proper HTTP(S) methods has been made, use of appropriate HTTP headers have been identified, HTTP status codes are used in responses.

The top level – level 3 – of the Richardson Maturity Model will not be reached. This third level assumes to make use of hypermedia controls: such controls allow REST servers to inform REST clients about the APIs that may be invoked in the current state of the application. While this is promising in terms of, for example, maintainability (e.g. APIs may be changed without having clients to understand these changes) no best practices have been established yet to deal with this.

2 Overall Approach

The following major steps should be followed to create a RESTful (level 2) API (see Figure 1).

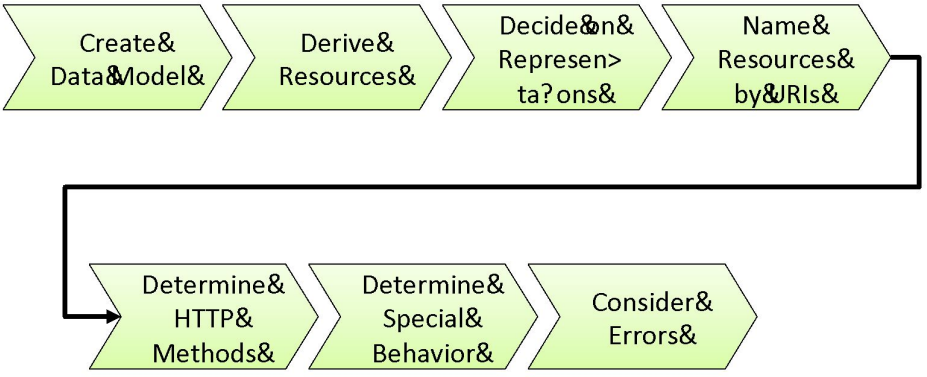

Figure 1 - REST API Design Approach

First, a data model of the data to be manipulated by the API is created. From this data model the resources of the API will be determined; typically, there is no one-to-one correspondence between data model elements and resources of the API, especially because new kinds of resources will typically be derived. For each of the resources the representations supported bay the API have to be determined. Next, these resources must be named properly by means of URIs. For each of the resources the HTTP methods used to

[1] Corresponding author – i.e. send all enquiries, comments,... to this mail address

perform the required application functions have to be decided; this does include the use of applicable HTTP headers. Special behavior required by the application (e.g. concurrency control, long running requests) has to be decided. Finally, potential error situations have to be identified and corresponding error messages have to be designed.

Note: Although Figure 1 sketches the approach as a sequential process, following it step-by-step is not always needed. For example,...

- ...the data model may already be known. In this case, the first step will be omitted.
- ...the resource model has already been decided. In this case, the second step may be left out. But in case the data model is not precisely specified, it may be worth to perform step 1.
- ...your API is very straight-forward, e.g. you don't expect concurrent updates of your resources, or none of your APIs kick-off long-running actions. Then you will leave out the step to "determine special behavior".

Each individual step of the overall approach will be detailed in the next sections.

3 Data Model

The data model behind an API can be specified by any *conceptual* data modeling language like the Entity-Relationship Model or UML Class Diagrams. In what follows we assume the use of the Entity-Relationship Model.

The main purpose of the data model behind an API is to specify the properties of the resources manipulated by an API in an *abstract* (i.e. implementation independent) manner. By specifying the attributes of the entity types of the data model no early decision is made about the format and media type in which instances of the entity types (aka *representations* in REST) are exchanged – this decision will be made later, and it can be changed during the lifetime of an API. Thus, it results in more flexibility in the development process.

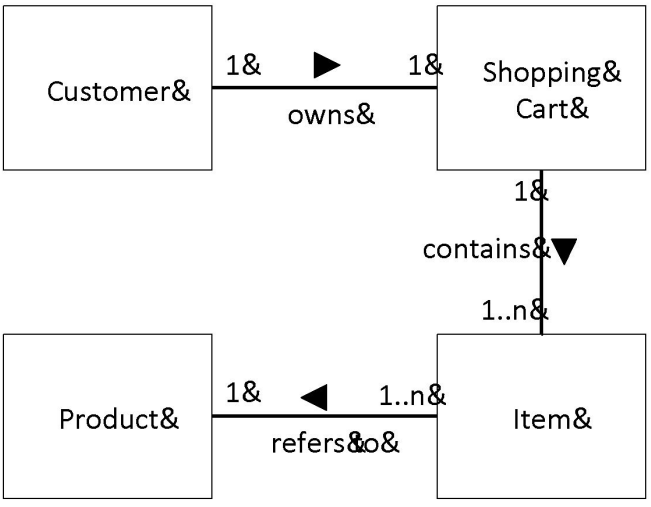

Figure 2 - Sample Data Model

Figure 2 depicts a sample data model that will be used in the following sections to give examples how to apply the given guidelines.

The data model is an important source to determine an appropriate resource model. However, the way clients interact with the API is a significant influencer of the resource model derived from the data model ("*clients win over data*"). The domain model drives the implementation of the API, while the resource model is driven by client interactions. But typically, the resource model will "*follow the data model*".

4 Resource Model

The resource model specifies the resources that are processed by the API. Several kinds of resources will be derived from both, the data model as well as the corresponding processing requirements.

4.1. Atomic Resources

The most basic decision to be made for deriving resources is identifying entities of the data model that are exchanged as a whole via the API. Such entities become *atomic resources*.

For example, based on the sample data model in Figure 2 the Customer entity will become such an atomic entity. This is because details about a customer like his address, payment information etc will be accessed in several scenarios supported by the API.

4.2. Collection Resources

The next decision to be made is whether atomic resources of the same type are needed to grouped into a set. Such bundles become *collection resources*.

For example, products will become a collection resource because the application supports a catalogue that allows browsing through (subsets of) all products available. Note, that there is no product***s*** entity type in the data model. Because the application requires such a collection resource we derive it from the data model and give it a new name that corresponds to the plural of the name of the grouped entity type.

As another example, item***s*** will become a collection resource that represent all items contained in a shopping cart of a certain customer. This collection will be scoped, i.e. only the items in a specific shopping cart are of interest but not the set of all items in all shopping carts (see section 5.6 for more details on scoping).

Finally, whenever the API supports the creation of an instance of one of the entity types of the data model, this entity type results in a corresponding collection resource. For example, a new customer may register with the application resulting in a new instance of the Customer entity type. Thus, Customer***s*** will become a collection resource.

4.3. Composite Resources

Sometimes, instances of groups of different entity types are manipulated as a whole because these instances are perceived as aggregates, e.g. they are typically collectively retrieved or deleted. Such groups become *composite resources*.

For example, a Shopping Cart is a composite resource because it is often retrieved or deleted as a whole, i.e. with all of its encompassed items.

4.4. Controller Resources

Controller resources are used when multiple resources have to be manipulated in a single API call in order to maintain data consistency. If integrity rules between resources must be obeyed, a client would have to understand these rules like the order in which resources are to be manipulated. By providing a controller resource to manipulate these resource in a single API call relieves the client from having to understand these rules – a significant contribution to loose coupling.

For example, deleting each individual item of a shopping cart one after the other may result in consistency problems in case an error occurs after having deleted only the first few items while others are still left in the shopping cart: a customer requesting the shopping cart just at this point in time of failure will realize a “broken” shopping cart.

Another example is the update of two account resources to realize a funds transfer – the classical motivation for ACID transactions. Each of these two accounts is an atomic resource, i.e. controller resources are different from composite resources.

4.5. Processing Function Resources

Processing function resources (aka *computing resources*) provide access to functions that either process particular resources, or that perform certain resource independent computations. In practice, processing function resources are often used for predefined partial updates of a resource.

For example, changing the status of a resource like the price of a product, or getting the official exchange rate between two currencies can be realized by means of a processing function resource.

Note: Partial updates are addressed by the HTTP PATCH method [7]. The problem with PATCH is twofold:

i. The PATCH method is not (yet) supported by all web servers. Of course, this problem may go away.

ii. The syntax and semantics for each use of the PATH method must be crisply defined:

    The resource enclosed in the body of a PATCH method is an *instruction document*, i.e. a set of instructions precisely describing what has to be updated and how, and all of these instructions must be performed atomically. Especially, the media type of this

instruction document is typically different from the media type of the resource that is to be modified by the PATCH. The instruction document may be perceived as a sort of transaction on the resource to be manipulated.

This results in broad exploitation of processing function resources for realizing partial updates.

4.6. Interpreting Relationships

One of the basic problems of deriving a resource model from a data model is in interpreting the relationships between entity types of the data model.

If manipulating instances of a certain entity type does not require the traversal of its associated relationships, then such an in entity type is a candidate for an atomic resource: if the instances have to be available via the API, the entity type is transformed into an atomic resource in a one-one correspondence.

Collection resources may be subject to an interpretation of its associated relationship types, i.e. collections may only make sense as children of other resources (so-called *scoped collections* – see 5.6)

For example: The Products collection resource is not scoped, i.e. this collection is a first class resource of the sample resource model. In contrast to this, the Items collection resource in fact is scoped: the collection of all items in all shopping carts is typically not of interest at all, but all items within a certain shopping cart is of interest. Thus, collections of Items dependent on a certain shopping cart is a collection resource (see section 5.6 how to denote such scoped collections).

If the API has to support the direct creation of instances of an entity type, this entity type results in a collection resource. This is because in the REST paradigm a collection resource is a factory for its members (see section 7.3). The entity type itself is the basis for atomic resources that are the members of the collection resource.

Processing function resources as well as controller resources result from functional requirements: they are typically not immediately derived from the data model but from update requirements or from requirements to derive information that may not even related to some other resources.

5 Resource URIs

The complete URI of an API complies to the following structure[2,3]:

```
{scheme}://{host}/{base-path}/{path}[?{query-string}]
```

The next sections will explain the elements of this structure.

[2] Strings in curly brackets {} represent variables that must be substituted by API-specific structures.
[3] Strings in square brackets [] represent optional elements.

5.1. Proper Naming

Proper naming of resources is key for an API to be easily understandable by clients. There are a few rules that should be followed ([2], [3], [5]):

- Atomic resources, collection resources and composite resources should be named as nouns because they represent “things”, not “actions” (actions would lean more towards verbs as names of resources).
- Processing function resources and controller resources should be named as verbs because they in fact represent “actions”.
- Processing function resources and controller resources should not be sub-resources of individual other resources.
    - I.e. they should not be names by means of a URI template (see 5.6).
    - Individual resources become parameters.
- Lower case characters should be used in names only because the rules about which URI element names are case sensitive and which are not may cause confusion.
- If multiple words are used to name a resource, these words should be separated by dashes (i.e. “-“).
    - Especially, no underscore (i.e. “_”) should be used: when names are rendered in browsers, they will be interpreted as links, i.e. shown underlined, and, thus, the underscore will be difficult to read.
    - Similarly, camel case or other programming language related naming should be avoided.
- Singular nouns should be used for naming atomic resources.
- Names of collections should be “pluralized”, i.e. named by the plural noun of the grouped concept (atomic resource or composite resource).
- Use forward slashes (i.e. “/”) to specify hierarchical relations between resource. A forward slash will separate the names of the hierarchically structured resources. The parent name will immediately precede the name of its immediate children.

5.2. Schemes

A *scheme* denotes the transport protocol supported by the API. Typically, WSO2 APIs will be all accessible over HTTPS, some APIs may support HTTP, some may support both schemes.

5.3. Host

The *host* part of the API specifies the domain of the API. For WSO2 hosted APIs, this value is apis.wso2.com. When hosted by or for customers this will be substituted by a customer-specific string.

5.4. Base Path

The *base-path* of an API follows the structure

```
/{feature-code}/[ {sub-code}/ ]/{version}
```

Thus, each base path consists of a feature-code structure indicating the feature for which this API is for. An optional sub-code structure may be used for features containing logically

independent collections of functionalities. For example, the feature-code may be “apim” for API Manager, in which case no sub-code is used, and the version may be v1.0 (see section 5.5 for details on versioning). For Enterprise Store, the feature-code may be “es”, and the independent Publisher functionality may get the “publisher” sub-code assigned.

The base path will be the same for all APIs of certain features or logically independent collections of feature functionalities, respectively. This base path will precede each proper resource name of the API, i.e. the path element of the API’s URI. Note, that a path is often encoded as a URI template (see section 5.6).

5.5. Versioning

The version of an API is specified as part of its URI. This version is specified as a pair of integers (separated by a dot) referred to as the major and the minor number of the version, preceded by the lower case letter “v”. E.g. a valid version string in the base path would be v2.1 indicating the first minor version of the second major version of the corresponding API.

Using this versioning scheme is referred to as *semantic versioning* [6]. In general, a version number following the semantic versioning concept has the structure major.minor.patch and the significance in terms of client impact is increasing from left to right:

- An incremented *patch* number means that the underlying modification to the API cannot even be noticed by a client – thus, the patch number is omitted from the version string. Only the internal implementation of the API has been changed while the signature of the API is unchanged. From the perspective of the API developer, a new patch number indicates a bug fix, a minor internal modification etc.
- An incremented *minor* number indicates that new features have been added to the API, but this addition must be backward compatible: the client can use the old API without failing. For example, the API may add new optional parameters or completely new request.
- An incremented *major* number signals changes that are not backward compatible: for example, new mandatory parameters have been added, former parameters have been dropped, or complete former requests are no longer available.

It is best practice to support the current major version as well as at least one major version back. In case new versions are released frequently (e.g. every few months) more major versions back have to be supported. Otherwise, clients will break (too fast).

When a client is using an API’s URI with a version number no longer supported, the server has to respond with the following response message that especially contains a Location header field with the URI of the latest version of the API:

```
HTTP/1.1 301 Moved Permanently
Location: ...
```

Note: there is a lot of debate on the subject of how to specify versions, and a couple alternative approaches to this area are presented. The discussion spans the whole spectrum from what the pure REST style considers a resource to what the big players providing Web APIs offer. What is recommended here is a pragmatic guideline.

5.6. URI Templates

A URI template is a URI certain elements of which contain strings in curly brackets [8]. These strings are *variables* that must be substituted by values when such a URI template is used by a client.

For example, when using the URI template

```
/shopping-carts/{shopping-cart-id}/{item-id}/product
```

“shopping-cart-id” and “item-id” are variables. These variable must be substituted when an API using this URI template in its path should be used.

A typical use of URI templates is in collection resources. An individual member of a collection will be identified by a unique value that will become the variable in the template immediately following the name of the corresponding collection resource. For example:

```
/products/{product-id}
```

Note the difference between this URI template and the one before. This URI template denotes a collection that is immediately derived from an entity type of the data model; all instances of this entity type should be accessible from the API, and they are grouped into the collection for that purpose.

The URI template before is different in the sense that it has been abstracted from a relation type of the data model, namely the relation between the Shopping Cart entity type and the Item entity type. Since there is no need to access the set of all instances of the Item entity type, no dedicated items collection type is part of the resource model. Instead, only items associated with a certain shopping cart (identified by its shopping-cart-id) should be accessible by the API, i.e. this set of items is scoped by the corresponding certain shopping cart: the collection of items is called a *scoped collection*.

5.7. Query String

By definition, an optional query string (if specified) is a part of the URI contributing to the unique identification of a resource. I.e. the URI without the query string and the URI with the query string identify different resources (a fact that is often ignored).

However, it is best practice to not use fields of the query string as identifier components. In this sense, a query string provides parameters to control the execution of the API processing the resource identified by the structure preceding the “?” symbol:

```
{scheme}://{host}/{base-path}/{path}?{query-string}
```

The query string consists of a sequence of name/value pairs that are separated by an “&”. The name-string and the value-string are separated by a “=”.

In practice, URIs are limited in size. Even worse, the maximum size supported by products differ. As a consequence, parameters of large query strings have to be moved into the message body of the corresponding requests. Note, that his will only work for requests that allow a message body (especially POST, but not GET).

6 Representation Specification

A format in which instances of the entity types of the data model are exchanged is referred to as a *representation* of such an instance. A representation is some sort of the shape of a resource, not the resource itself. In this sense, a representation is some sort of view onto the resource.

The information content of an atomic resource or a composite resource is immediately defined by the data model underlying an API. The values of the attributes and – if appropriate – the identifiers of associated resources make up the information content of an atomic resource. Similarly, the aggregate of the information contents of the resources of a composite resource is the information content of the composite resource.

A data structure must be decided for this information content. In addition, one or more renderings of this data structure must be decided. For example, a certain data structure may be rendered as a JSON document or an XML document. Typically, renderings of data structures are specified by means of MIME types. A specific rendering of a data structure is referred to as the *representation* of the resource, i.e. a representation has a MIME type. Keep in mind, that this MIME type does not indicate the data structure of the information exchanged between the client and the API implementation but only the rendering.

Note: In most practical situations, a single representation (e.g. JSON) for all entities exchanged via an API suffice.

7 HTTP Methods Used

Manipulation of resources in the REST style is done by create, retrieve, update, and delete operations (so-called CRUD operations), that mapped to the HTTP methods POST, GET, PUT, and DELETE.

A request that can be used without producing any side-effect is called a *safe* request. A request that can be used multiple times and that is always producing the same effect as the first invocation is called *idempotent*.

7.1. GET

GET is in HTTP as well as in the REST style specified as a safe and idempotent request. Thus, an API using the GET method must not produce any side-effects. Retrieving an atomic resource (4.1) or a composite resource (4.3) is done by performing a GET on the URI identifying the resource to be retrieved.

Retrieving a (subset of) resources of a certain type is done by performing a GET on the URI of the collection resource (4.2) of that type, and specifying a filter condition (10.2).

7.2. PUT

PUT substitutes the resource identified by the URI. Thus, the body of the corresponding PUT message provides the modified but complete representation of a resource that will completely substitute the existing resource: parts of the resource that have not changed must be included in the modified resource of the message body. Especially, a PUT request must not be used for a partial update. As a consequence, PUT is an idempotent request (but not safe).

Partial updates, i.e. updates that modify only selective pieces of an existing resource have to be realized by means of corresponding processing function resources (see section 7.3).

Note: PUT may be used to create a new resource, but this is not recommended. The reason is that in this case, the client is in charge of creating such a globally unique URI identifying the newly created resource – and ensuring uniqueness of identified is a difficult task. In contrast, using POST to create new resources reliefs the client for creating unique identifiers because the server will create the URI and return it to the client (see section 7.3)

7.3. POST

POST is neither safe nor idempotent. The main usages of POST are the creation of new resource, and the initiation of functions, i.e. to interact with processing function resources (4.5) as well as controller resources (4.4).

In order to create a new resource, a POST request is used with the URI of the collection resource to which the new resource should be added. If the POST is processed successful, the response message will especially include a Location header that will have the newly created URI of the added resource as value. Also, it is good practice to return in the response message a Last-Modified header containing the time the resource has been created, as well as the ETag header containing the entity tag of the new resource.

It is often appropriate that the client checks the correctness of the created resource. For this purpose, the response message body contains the resource as it would be returned by retrieving it from the URI of the Location header. In this case, the response message also includes a Content-Location header repeating the URI of the newly created resource.

7.4. DELETE

A resource is deleted by means of the DELETE request on the URI of the resource. Once a DELETE request returned successfully with a “200 OK” response, following DELETE requests on the same URI will result in a “404 Not Found” response because there is no resource available with the URI of the deleted resource.

Note: By definition, DELETE is an idempotent request, which has the following curious theoretical implication. Responding with “200 OK” to the first DELETE and with “404 Not Found” for any further DELETE on the same URI is not quite RESTful because the responses of the first and all further requests are different; thus, the request is not idempotent. In order to fully comply to the REST style, a server would have to maintain all URIs of deleted

resources in order to always respond with “200 OK”, i.e. with the same response. This is considered to much of an effort for nearly no gain, i.e. this is not implemented in practice.

8 Headers

HTTP headers provide the vehicle for many non-functional properties of REST APIs. The following list of HTTP headers are used in most APIs.

8.1. Request Headers

Accept
This is the list of content types acceptable for the client.

Authorization
The credentials of the client for authentication by the server.

Content-Type
This is the MIME-type of the message body of the PUT or POST request.

If-Match
Used to avoid concurrency conflicts: if the client-passed entity tag is identical to the entity tag of the resource at the server-side, the request if performed.

If-Modified-Since
Used to avoid retrieving data that has been cached by the client. If the client-provided timestamp is identical to the time the entity has been modified last at the server side no message body is returned.

If-None-Match
Used to avoid retrieving data that has been cached by the client. If the client-provided entity tag is identical to the entity tag of the resource at the server side no message body is returned.

If-Unmodified-Since
Used to avoid concurrency conflicts. if the client-passed last-modified time stamp is identical to the time the resource has been changed last at the server-side, the request if performed.

8.2. Response Headers

Content-Location
The URL of the message body. For example, the URL of the resource describing the status of a long running request (see 10.6).

Content-Type
The MIME-type of the message body.

ETag

A “fingerprint” of the resource as currently available at the server, often a digest of the resource.

Last-Modified
The timestamp when the resource has been modified the last time at the server.

Location
The URL of a newly created resource.

WWW-Authenticate
An indication of the authentication scheme to be used to access the resource.

9 Status Codes

HTTP status codes [9] are returned by response messages and provide key information to clients about the status of a request. The following status codes are used in many APIs.

200 OK
The request has been performed successfully. If the request was a GET, the requested resource is returned in the message body. If the request was a POST, the result of the requested action is described by the message body, or it is contained in the message body.

201 Created
The request has been performed successfully. The URL of the newly created entity is contained in the Location header of the response. An ETag header should be returned with the current entity tag of the resource just created. The response may also contain an entity corresponding to the created resource.

202 Accepted
The processing of the request has started but will take some time (see 10.6). The success of the processing is not guaranteed and should be checked by the client. The body of the response message should provide information about the current state of the processing, as well as information about where the client can request updated status information at a later point in time; typically, the Content-Location header of the response contains a URL where this status information can be retrieved via GET.

303 See Other
The response of the request is available at a different URL; this URL is given as value of the Location header of the response message. Typically, this status code is returned after the processing of a long running request is completed and the client retrieves the status of the long running request (see 10.6).

304 Not Modified
The requesting client has already the latest version of the requested resource. Thus, the body of the response message must be empty. This status code is returned as a result of a conditional GET (see 10.4), and the the specified conditions (i.e. If-Non-Match, If-Modified-Since) are not met.

400 Bad Request
The request is invalid. For example, syntax errors in expressions passed with the request are found, values are out of range, required data is missing etc.

401 Unauthorized
The request requires client authorization or the passed credentials are not accepted. The response must include a WWW-Authenticate header. The request may be repeated by the client including proper credentials in the Authorization headers.

403 Forbidden
The server understood the request but refused to perform it. For example, the request must be conditional but no condition has been specified.

404 Not Found
The requested entity does not exist.

406 Not Acceptable
The requested media type is not supported. For example, a GET request wants to retrieve an entity in a media type (specified as value of the Accept header of the request) not supported by the server.

412 Precondition Failed
The request has not been performed because one of the preconditions has not been met. This status code is returned when the request was conditional (see 10.5) and one of the conditions specified (i.e. If-Match, If-Unmodified-Since) is not met.

415 Unsupported Media Type
The entity passed by the request was in a not supported format. For example, a PUT request passed an entity in its body, and this entity was in a format or media type, respectively, that is not understood by the server.

Note: 5xx Status codes denote severe errors at the server side or the network, or denote not implemented functions etc. Such errors are very generic, i.e. there is no need to document them explicitly for an API.

10 Special Behavior

Except for very simple APIs, a REST API has to offer features that allows to cope with advanced situations like large result sets, concurrent updates, or long running request. The following describes best practices to deal with some of those special situations.

Note: It is good practice for an API to support at least queries (see 10.2) and pagination (see 10.3). For example, this will allow to support push-down of filtering etc. from an API orchestration (a more and more importance API technology [10]) to the individual APIs as optimization of response time, bandwidth usage etc.

10.1. Content Negotiation

The REST style clearly distinguishes between a resource itself (i.e. as an abstract entity) and its different possible representations. Such a *representation* is a rendering of the resource's information content in the format of a certain media type. Which of the representation of a resource is returned as content of the message body is negotiated between a client and the resource provider (a.k.a. *content negotiation*).

Server-driven content negotiation assumes that a server processing a request determines the representation best suited to the requesting client. The server is dependent on the requesting client to specify information about its processing capabilities or requirements. The latter is done by means of header fields of the request message like Accept, Accept-Encoding etc.

In *client-driven content negotiation* the server detects that it has more than one representation of a resource available that may serve the client's needs. The server responds to the request with a "300 Multiple Choices" message that carries information about the representations available, and the client finally selects one of these representations and explicitly requests it in a following request.

Server-driven content negotiation has the advantage of avoiding a second request to be made by the client, but has the disadvantage of potentially responding a representation to the client that is not ideally suited. Client-driven content negotiation has the advantage that the client gets the representation that is best suited for it, but the disadvantage of two round-trips.

In the following request, the client specifies that it is able to process JSON, XML as well as plain text, but that it prefers JSON. The preferences in media types is specified by weighting a media type by a quality value q. The server will determine which of the representations it has available and will return the one with the highest quality value.

Example:

```
GET /products/27182
Accept: application/json;q=0.9, application/xml;q=0.6, text/plain;q=0.1
...
```

In case a client specified representations the server has not available, the server will respond with a message returning a corresponding status code:

```
HTTP/1.1 406 Not acceptable
```

Note: Even in case an API documents that it supports only one certain media type, a client may pass a request that contains an Accept header with a different media type. In this case, the API implementation must return the "406 Not acceptable" message.

10.2. Queries

A query on a collection resource consists of three artifacts: (i) a mandatory filter condition, (ii) an optional sort expression, and (iii) an optional projection. First, a list of attributes from the entity type on which the collection is based on has to be distinguished that can be use in either of these artifacts.

A *filter condition* is Boolean expression in these attributes. The spectrum of filter conditions is from simple (supporting to specify a single attribute name with a value being compared for equality) to complex (arbitrary conditions and arbitrary comparison operators).

Example of a complex query:

```
filter=((price > 1000 AND status = on-stock) OR (price < 200 AND NOT(status = on-stock)))
```

A *sort expression* consists of a list of attribute names together with the indication for each attribute name whether the result is to be sorted ascending or descending with respect to the corresponding attribute. The order in which the result is sorted is implied by the order of the list of attribute names.

Example:

```
sort=(price ASC, delivery-date DESC)
```

A list of attribute names that have to be used to create each item in the result set is called a *projection*. Each specified attribute name is used to extract the corresponding information from each resource qualified and compile a corresponding item in the result set.

Example:

```
projection=price,color,status
```

There are two ways to enable queries on collections. First, a query is part of the query string of the URI used by a GET. Second, the query is pass in the body of a POST request of a special processing function resource.

An example of specifying a query as query string of the URI is:

```
GET /products?status=on-stock&sortAsc=price&projection=price,color,status HTTP/1.1
```

Specifying a query as part of a query string concatenated to the URI of the collection resource that is enquired is the preferred way for queries. This is because a GET is used with this URI, clearly expressing the semantics of the request namely retrieving a subset of the collection. However, in practice URIs have a maximum length depending on the browser or Web server used. When queries may become thus complex that the maximum length may be exceeded, the use of a POST request that contains the (complex) query in the request body is enforced. For this purpose, a separate processing function resource is to be realized:

```
POST /product-search HTTP/1.1
...

filter=((price > 1000 AND status = on-stock) OR (price < 200 AND NOT(status = on-stock)))
sort=(price ASC, delivery-date DESC)
projection=price,color,status
```

10.3. Pagination

When a large collection resource (or subsets of it) is to be retrieved, it is often convenient to retrieve the result set in smaller chunks: for example, the latency of the request is reduced, clients can predict the amount data to be dealt with etc.

For this purpose, the retrieval request specifies a query string containing an “offset” field as well as a “limit” field; the offset is the position number of a qualified resource where the retrieval should start, and the limit is the maximum number of resource to be returned.

The response message with the subset of the qualified resource returned should specify the total number of all qualified resources (“count” field), a link to next chunk of qualified resources (“next” field), as well as a link to the previous chunk of qualified resources (“previous” field). The actual format how these fields as well as the set of resources is returned, is application specific, i.e. the following example is intended to show the principle only.

Example:

```
GET /products?offset=42&limit=3
...
```

Response:

```
HTTP/1.1 200 OK

count=119
next={link-to-next-subset}
previous={link-to-previous-subset}
P42-details
P43-details
P44-details
```

10.4. Client-Side Caching

A client may cache resources as well as header fields of resources to reduce transfer of data that has not been changed since its last retrieval. For this purpose, a client uses a conditional request to retrieve data. Such a conditional request specifies the If-None-Match or the If-Modified-Since headers in the request.

The value of the If-None-Match header is the value of the ETag (Entity Tag) header of the resource as retrieved last time by the requesting client. The value of the If-Modified-Since header is the value of the Last-Modified header of the resource as retrieved last time by the requesting client. When performing the conditional request, the API implementation has to compare the value(s) passed by the client in the request with the corresponding values of the current resource as stored by the server (note[4] that If-None-Match takes precedence over If-Modified-Since). If the values have not changed, the resource is not returned (response code “304 Not Modified”); otherwise, the resource is returned.

Example:

```
GET /products/31415
```

[4] http://tools.ietf.org/html/rfc7232#section-6

```
If-None-Match: “42049dcaf450987cffd”
If-Modified-Since: Thu, 7 Jan 2016 17:41 CET
```

Response:

```
HTTP/1.1 304 Not Modified
```

10.5. Concurrency Control

Multiple clients interacting with the same resource may cause concurrency conflicts like lost updates. *Pessimistic concurrency control* mechanisms avoid conflicts in advance by locking resources. This is a good choice, if the probability for conflicts is high. *Optimistic concurrency control* avoids conflicts by detecting conflicts and signaling them to clients that may retry their requests. This is a good choice, if the probability for conflicts is low.

Many concurrent interactions in REST-based APIs can be handled by optimistic concurrency control. For this purpose, requests are send as conditional requests. A *conditional request* specifies the If-Match or If-Unmodified-Since header in the request. The value of the If-Match header is the value of the ETag (Entity Tag) header of the resource as retrieved last time by the requesting client. The value of the If-Unmodified-Since header is the value of the Last-Modified header of the resource as retrieved last time by the requesting client. When performing the conditional request, the API implementation has to compare the value(s) passed by the client in the request with the corresponding values of the current resource as stored by the origin server (note[5] that If-Match takes precedence over If-Unmodified-Since). If the values have been changed, the request is rejected (response code “412 Precondition Failed”); otherwise, the request is executed.

Example:

```
PUT /products/31415
If-Match: “42049dcaf450987cffd”
If-Unmodified-Since: Thu, 7 Jan 2016 17:41 CET
```

Response:

```
HTTP/1.1 412 Precondition Failed
```

In order to send conditional requests, the requesting client either has to cache these values after having retrieved the resource before, or has to retrieve these values by a GET on the resource that is performed before the conditional request is made. The disadvantage of the latter is that a second request has to be made.

Example:

```
GET /products/31415
...
```

Response:

```
HTTP/1.1 200 OK
ETag: “4562aae7732a56”
Last-Modified: Wed, 6 Jan 2016 11:13 CET
```

[5] http://tools.ietf.org/html/rfc7232#section-6

```
...
```

Even with PATCH, special care must be taken with respect to lost updates: if concurrent PATCHes are to be supported, concurrency control has to be implemented as a conditional request.

The similar pattern may be used to realize partial updates without using processing function resources (section 4.5): A client has to GET the resource to be updated, apply the partial updates locally, and then send a conditional PUT with the complete resource to the server.

10.6. Long Running Requests

Requests may take too long to return the response synchronously, i.e. such requests have to be processed asynchronously. For example, a request like applying for a credit (see next example) may kick-off a workflow involving human beings that will take some time. In this case, the API will accept the request and return both, the URI of a so-called task resource as well as the task resource itself in the response body. The task resource is a document like in the following example, that contains the actual status of the long running request. The URI of the task resource is passed in the Content-Location header field of the response. This URI can be used by the client to poll for the processing state of the long running task later on.

Example:

```
POST /apply-for-credit HTTP/1.1
...
```

Response:

```
HTTP/1.1 202 Accepted
Content-Type: application/xml
Content-Location: http://www.shark-credits.com/apply/tasks/1

<status>
 <state>running</state>
 <link rel="self" href=".../tasks/1"/>
 <estimatedCompletion>2020-04-01</estimatedCompletion>
</status>
```

The response message specifies by the “202 Accepted” status code that the request is accepted but will be processed asynchronously. The task resource says that the request is running and gives an estimated completion time, amongst other appropriate information (note, that there is no standardized format of a task resource). In succeeding requests, the client will retrieve the actual task resource via the URI value of the Content-Location header field:

```
GET /apply/tasks/1 HTTP/1.1
Host: www.shark-credits.com
...
```

The sample response succeeds successfully (“200 OK”), which means that the retrieval of the task resource was successful – note especially that this does not indicate that the long running request completed successfully:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Location: http://www.shark-credits.com/apply/tasks/1

<status>
 <state>running</state>
 <link rel="self" href=".../tasks/1"/>
 <estimatedCompletion>2020-10-10</estimatedCompletion>
</status>
```

After some time, the long running request will have completed, i.e. the GET request above will receive the following response:

```
HTTP/1.1 303 See Other
Content-Type: application/xml
Location: http://www.shark-credits.com/apply-for-credit
Content-Location: http://www.shark-credits.com/apply/tasks/1

<status>
 <state>ready</state>
 <link rel="self" href=".../tasks/1"/>
 <message>Image processed & stored</message>
</status>
```

The status code “303 See Other” specifies that a new resource has been created the URI of which is the value of the Location header field. This is the URI of the result of the long running request. The task resource now reports the in its state field that the long request succeeded successfully.

It may happen that the long running request fails. In this case, the retrieval of the task resource will succeed with a “200 OK” status code and the task resource’s status will report that the long running request completed but not successfully. Thus, it must be kept in mind, that the status code of the response of the retrieval of the task resource is not related to the status of the long running request at all.

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Location: http://www.shark-credits.com/apply/tasks/1

<status>
 <state>FAILED</state>
 <link rel="self" href=".../tasks/1"/>
 <estimatedCompletion>2020-10-10</estimatedCompletion>
</status>
```

11 Reporting Errors

When a 4xx status code is returned, a client error has been detected. In this case, more detailed error information should be returned in the body of the response message. The following error information (in YAML) detailing the client error will help the client to understand how to fix the request and repeat it successfully.

```
Error:
  title: Error object returned with 4XX HTTP status code
  required:
    - code
    - message
  properties:
    code:
      type: integer
      format: int64
      description: |
        A product-specific error code.
    message:
      type: string
      description: |
       A detailed description of the error occured.
    description:
      type: string
      description: |
        A short description about the error message.
    moreInfo:
      type: string
      description: |
        Preferably a URL with more details about the error.
    error:
      type: array
      description: |
        If more than one error occurred they are listed separately.
        For example, list out validation errors by each field.
      items:
        $ref: '#/definitions/ErrorListItem'
```

If more than one error occurred, each error is reported as an element of the following array that is part of the error object before.

```
ErrorListItem:
  title: Description of individual errors that may have occurred during a request.
  required:
    - code
    - message
  properties:
    code:
```

```
    type: integer
    format: int64
  message:
    type: string
    description: |
      Description about the individual error occurred
```

12 Security

Access to resources of an API are typically secured. These resources should be accessible based on a properly designed permission model to prevent misuse of APIs. For example, adding, updating, or deleting permissions to access resources or operations as well as associated users or roles etc. is needed. Furthermore, access requirements can change, i.e. the ability to manage these permissions flexibly is required.

The following examples elucidate this:

- The user who created a Shopping Cart has the permission to create and delete an individual Item from the shopping cart.
- Users need explicit permissions to add new Products or update existing Product information.
- A different set of users will get permission to delete Customers (for example those with bad credit standing).

Thus, different resources are protected by different permissions and these permissions may change. To support this, REST APIs need to support extensible security mechanisms like HTTP Basic Authentication, OAuth, or XACML.

Note: Because of the complexity of XACML, OAuth is the pragmatic way of adding security to APIs. In what follows, XACML is sketched for completeness reasons only.

12.1. Basis Authentication

HTTP Basic Authentication [11] requires that a request contains an Authorization header field containing user-id and password credentials in Base64 encoding:

```
Authorization: Basic Zm9d03wWUdz==
```

In case the request does not contain the credentials, the server will respond with a 401 status code containing the information about the protected resources (“realm”) and the name of the authentication scheme to be used (i.e. “Basic”):

```
HTTP/1.1 401 Unauthorized
...
WWW-Authenticate: Basic realm="/product"
...
```

This information will be used by the client to produce the correct value of the Authorization header to be passed with the repeated request. Otherwise, the request will fail again.

Note, that HTTP Basic authentication is unsecure. This is because the credentials are in clear text, i.e. it is Base64 encoded, which can be immediately translated into clear text. Because of this, HTTP Basis authentication is only viable over HTTPS. HTTP Digest encoding is more secure but is subject to a “man-in-the-middle-attack”. By using HTTPS, this is avoided. Thus, HTTP security mechanisms are only acceptable for non-critical APIs or for requests that generate more secure access tokens (see next) – or these mechanisms are used over HTTPS only.

12.2. Open Authorization (OAuth)

The likelihood of possible attacks of HTTP Basic or Digest authentication is reduced by using OAuth [12] tokens with the HTTP Bearer authentication mechanism. The request message contains an Authentication header with the OAuth access token:

```
Authorization: Bearer mF_9.B5f-4.1JqM,
                    scope="API-Creation ..."
 ...
```

The string after the Bearer keyword is the credential provided by means of OAuth protocols and mechanisms. The value of the scope field is a list of strings denoting the resources intended to be accessed by the request. In case the token provided is not sufficient for the scopes, the request fails.

Note: The use of OAuth protection of APIs requires a design of scopes along with the APIs. A set of scopes needs to be designed that protects individual processing function resources, (subsets of) collection resources, or the ability to create or delete resource, for example.

While OAuth reduces the likelihood of attacks when used over HTTP, there are several attacks known to OAuth. Thus, although OAuth has been proposed as a mechanism over HTTP, it should be used over HTTPS to increase security.

12.3. eXtensible Access Control Markup Language (XACML)

A more fine-grained control of access to resources (i.e. finer than scopes) is supported by XACML [13], but XACML is considered quite complex in practice. The following sketch should underpin this.

XACML is an attribute-based access control (ABAC) mechanism, in contrast to role-based access control (RBAC) mechanisms. As such, access to a resource is determined based on attributes contained in a request message.

For this purpose, XACML describes an access control policy language, a request/response language and protocol, and a reference architecture. The policy language is used to express access control policies (who can do what in which context). The request/response language

supports queries about whether a particular access should be allowed (requests) and specifies answers to those queries (responses). The reference architecture proposes deployment of necessary components within an infrastructure to allow efficient enforcement of policies.

XACML Processing

When using XACML to secure API access, a policy enforcement point (PEP) is assumed that is in charge for extracting the required parameters from the API request message and validate permissions by a policy decision point (PDP) based on these parameters. The PEP will create a XACML request based on the extracted parameters. This request is send to the PDP to validate permissions. The result of permission validation is returned by the PDP in the response message and is based on predefined policies. The response message will contain one of the following status:

- Permit – the access is granted.
- Deny – the access is not granted.
- Indeterminate – an error occurred, or some required data was missing and no decision could be made.
- Not Applicable – the request could not be processed.

Implications on API Design

When an API call is received, the PEP must return an HTTP 401 Unauthorized status code if anything other than Permit results from access validation. In case the request is authorized (i.e. Permit resulted from access validation), the request will be pass to the API implementation.

XACML may be used with other authentication or authorization mechanisms such as HTTP Basic Authorization or OAuth.

Note: The use of XACML for protecting APIs requires (i) an implementation that follows the architecture above, and (ii) the ability for API administrators to define corresponding policies.

13 References

[1] http://martinfowler.com/articles/richardsonMaturityModel.html
[2] M. Masse: *REST API – Design Rulebook*. O'Reilly 2012.
[3] S. Allamaraju: *RESTful Web Services Cookbook*. O'Reilly 2010.
[4] J. Webber, S. Parastatidis, I. Robinson: *REST in Practice*. O'Reilly 2010.
[5] https://github.com/tfredrich/RestApiTutorial.com/blob/master/media/RESTful%20Best%20Practices-v1_2.pdf
[6] https://en.wikipedia.org/wiki/Software_versioning
[7] https://tools.ietf.org/html/rfc5789
[8] http://tools.ietf.org/html/rfc6570
[9] http://www.restapitutorial.com/httpstatuscodes.html
[10] http://www.danieljacobson.com/blog/306
[11] https://tools.ietf.org/html/rfc2617
[12] https://tools.ietf.org/html/rfc6749
[13] http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.pdf